1998

Lazy Algorithms in Parallel Discrete Event Simulation

Reuben Pasquini

Report Number:
98-020

LAZY ALGORITHMS IN PARALLEL DISCRETE EVENT SIMULATION

Reuben Pasquini
Vernon Rego

Purdue University
Department of Computer Science
West Lafayette, IN 47907

CSD-TR #98-020
June 1998

Lazy Algorithms in Parallel Discrete Event Simulation

Reuben Pasquini
pasquini@cs.purdue.edu

Vernon Rego
rego@cs.purdue.edu

June 24, 1998

Abstract

Optimistic parallel discrete event simulation (PDES) systems rely upon the time warp synchronization algorithm, or some variant, to enforce causality. In a parallel simulation, causality is violated when one processor sends another processor a *straggler* event which is scheduled to execute at a virtual time (v.t.) in the latter's past. When a processor receives a straggler event, the processor rolls its local state back to a state saved at a v.t. preceding the straggler's scheduled v.t. We address the overhead of a processor's roll back from its current time to a past time. We propose new lazy techniques for saving state, calendar management, and gvt scheduling, to enhance system performance. We present our results in the context of the ParaSol system.

1 Introduction

The problem of rapidly simulating large, complex systems promises to be one of the major challenges of the next decade. For example, the next generation of computer chips will have in excess of one billion transistors making up hundreds of millions of logic gates [10], metropolitan air-traffic control systems must safely manage time and space constraints for hundreds of aircraft every hour [34], and modern military engagement requires the coordination of thousands of entities including aircraft, ships, soldiers, commanders, satellites, computers, and communication systems. Simulators have traditionally aided in the task of understanding, designing, building and operating such systems.

Discrete event simulation (DES) is a technique that exploits a computer to model a system whose state changes (stochastically) at discrete points in time. A simulation program operates on a model's *state* variables during each of a sequence of time-ordered *events*. It is not enough for a simulator to process events quickly; a simulation language must also export an API with which large models may be simply specified and modified. Parallel discrete event simulation (PDES) algorithms attempt to speed up the execution of a DES by distributing simulation workload across distinct processors. We believe that PDES offers great promise for meeting the simulation needs of developers of increasingly complex systems.

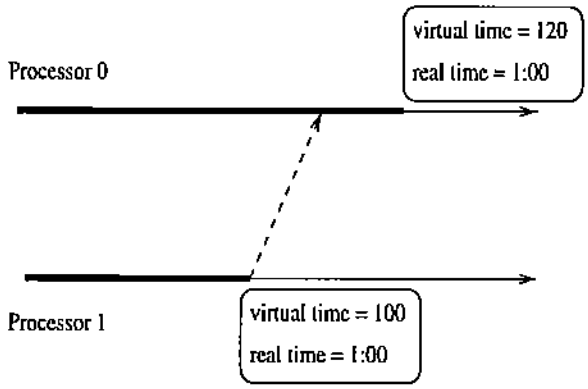
A *Parallel Simulation Language* (PSL) enables a user to specify a PDES program for execution on a multiprocessor. From the standpoint of PSL design, there are two main goals. First, a PSL's runtime system should yield speedup, relative to the runtime system of a comparable sequential simulation language (SSL). Ideally, given sufficient concurrency in the model, speedup should scale with the number of processors employed. Second, a PSL should provide the requisite simulation functionality while hiding the complexity (e.g., message passing, synchronization, state-saving) of parallel simulation. That is, the user should be given the expressive power of a SSL with little or no additional complexity.

1.1 Parallel Simulation Concepts

PDES algorithms attempt to speed up the execution of a DES program by distributing simulation workload across processors. A DES executes a time-ordered sequence of simulation *events*. Each event may access one or more simulation objects and schedule one or more future events. The state of the simulated system is defined by the state of all simulation objects. The order in which events execute is determined by a *virtual time* which, in turn, is defined by event time–stamps. Events execute in nondecreasing time–stamp order so that *virtual time* always advances.

It is natural to think of parallelizing DES programs by distributing all simulation events across processors. Given n processors and m events, each processor would ideally handle m/n events, suggesting an ideal speedup of n. Unfortunately, distributed events typically don't access simulation objects in time–stamp order. For example, processor P_1 may execute an event e_1 with time–stamp $t_1 = 1$ after processor P_2 executes an event e_2 with time–stamp $t_2 = 2$. If e_2 happens to access a shared simulation object (i.e., an object shared by P_1 and P_2) before e_1 is able to access the object (say, because of processor or network delay), then the parallel execution witnesses e_1 and e_2 access the shared object in an order that is different from the order in which these events access the object in a sequential execution (e_1 followed by e_2).

A PDES must employ an algorithm which ensures that events are executed in a *causally consistent* way. A simulation is causally consistent if each simulation object is accessed by events in nondecreasing time–stamp order. In seminal works on achieving causal consistency, Chandy and Misra [12] and Jefferson [20] came up with very different solutions. The Chandry-Misra algorithm avoids causality errors by ensuring that each processor executes events in time-stamp order. For this, each processor P *conservatively* executes event e_j with time-stamp t_j only when P is certain



Processor P_1 generates an event for processor P_0 at virtual time 100, but P_0 has already reached v.t. 120. Processor P_0 will roll back when P_0 receives the straggler event.

Figure 1: Causality error in an optimistic parallel simulation

that no event e_i with an earlier time-stamp $t_i < t_j$ will be scheduled on P (by another processor). Because of this, the Chandy-Misra algorithm is called a *conservative* parallel simulation algorithm [12]. Research in conservative parallel simulation has focused on the problem of deadlock avoidance. Deadlock can arise when each of a cycle of processors blocks, awaiting input that will allow the processor to continue simulating without spoiling causality. Sophisticated synchronization and lookahead algorithms have been proposed to address the deadlock problem [32, 15, 8, 7].

The time warp algorithm is an *optimistic* algorithm for PDES. It is optimistic in the sense that each processor executes events in time–stamp order under the optimistic assumption that causality is not being violated. At any point, however, a processor may receive an event (from another processor) whose time–stamp indicates that it should already have been processed; such an event is called a *straggler*. On detecting a straggler, a processor *rolls back* (i.e., a rollback) to a checkpointed system state that corresponds to a time–stamp which is less than the straggler's time–stamp. Execution continues from this point, and the straggler is processed in the right time–stamp order (see Figure 1). Research in optimistic parallel simulation has focused on reducing runtime overheads: state-saving [28, 39, 19, 17, 37, 24], rollback [36, 4, 30, 1], and global virtual time (gvt) computation [5, 11, 31].

1.2 Outline

We propose new lazy approaches to state saving, calendar management, and gvt scheduling which try to avoid doing work which is thrown away as the result of rollbacks in optimistic PDES. In Section 2 we introduce existing PDES tools, including the ParaSol system that we use in our research. In Section 3 we describe a new lazy object-based state saving algorithm. We present the results of experiments comparing our lazy algorithm with other approaches to state saving. In Section 4 we introduce a new lazy hybrid calendar and compare its performance with other calendar implementations. In Section 5 we present some of the techniques we use to allow our lazy algorithms to fully exploit the lookahead present in a simulation, and in Section 6 we present a lazy algorithm for scheduling gvt computation. We conclude briefly in Section 7.

2 PDES Systems

The Time Warp Operating System (TWOS) was an early implementation of the time warp mechanism [36, 35]. TWOS is an event-based system in which logical processes (LPs) exchange time-stamped event messages. TWOS was designed to run on bare hardware, with a bottom layer responsible for low-level tasks (context management, message communication, interrupt handling, etc.)

and a top layer responsible for time warp mechanisms (rollback, anti-messages, etc.). SPEEDES [40, 41, 39] is a direct descendant of TWOS; it is written in C++, with an object-oriented view. Like TWOS, SPEEDES is event-based. It supports multiple synchronization protocols and a delta exchange mechanism [41] for incremental state-saving [39].

The *Georgia Tech Time Warp* (*GTW*) is a library supporting the construction of event based parallel discrete event simulations [18, 33]. *GTW* is available for various shared memory parallel machines, and a new version of *GTW* which runs over distributed memory architectures (including a network of workstations) using *PVM* is under development. *GTW* supports the event-scheduling world view. The simulation developer defines events which the simulation passes between processor's as messages. *GTW* also relies upon the developer to supply initialization and state saving routines for each processor making model specification in *GTW* more difficult than specification in a similar sequential language.

Maisie is a parallel simulation language which supports the *active server* world view. *Maisie* specifies models as a set of entities which communicate by exchanging typed messages. *Maisie* improves upon *GTW* by making constructs for parallel execution more transparent to the user. Using *Maisie* we can specify a model for sequential execution without constructs for parallel execution, then later add constructs supporting parallel execution if the sequential model does not yield a satisfactory runtime [3].

SIMKIT is a PDES language extending C++ under development at the University of Calgary. *SIMKIT* exploits object oriented design techniques to support an active-server world view. *SIMKIT*'s object oriented architecture is built upon a simulation kernel which manages the simulation calendar and time warp algorithm. Simulation developers may build domain specific libraries which provide state saved objects for different types of models [2]. *SIMKIT* supports the active server world view and runs on shared memory architectures.

The *APOSTLE* system employs the *breathing time-buckets* algorithm to enforce the causality constraint [43, 6]. This algorithm constrains its optimism by periodically synchronizing the processors participating in the simulation. Periodic synchronization may help decrease the frequency of rollback [41].

We use the ParaSol PDES system in our experiments. ParaSol provides a simulation developer with libraries of C++ classes with which he can instantiate simulation processes (migratable threads) and objects. ParaSol's layered architecture hides state saving and interprocessor communication code from the user who describes his models using ParaSol's programming interface. ParaSol's API is an object oriented derivative of the API for the sequential simulation language, *CSIM* [21, 27, 38].

3 Lazy Checkpointing

Saving state is a large runtime overhead of the time warp simulation algorithm. Each processor participating in a parallel simulation periodically saves a snapshot of its local state. When a processor receives a straggler event, the processor rolls back to a state with virtual time (v.t.) preceding the v.t. at which the straggler is scheduled to execute. When a processor rolls back to a v.t. t_{rb}, the processor restores its local simulation state from a snapshot of the simulation state at t_{rb}.

Saving state is pure overhead from a simulator's event-processing viewpoint. Since a sequential simulation does not save state, state saving cost limits the speedup a parallel simulation can achieve. For example, suppose that after each event e a parallel simulation must save the state of an object x_e. If the time spent to execute e is t_e, and the time required to save object x_e is t_x, then the parallel simulation requires at least $(t_e + t_x)/t_e$ processors to break even (i.e., yield a speedup 1.0)

	sample from exponential		memcpy 100 bytes	
	mean	variance	mean	variance
IBM Thinkpad, Linux 2.0.x	52.12	1.82	7.97	1.28
Sun Sparc 5, Solaris 2.5.1	179.66	12.75	35.93	0.4294

(a) We collected this data by averaging over 100 samples the runtime (collected via the UNIX `times()` system call) needed to perform 200000 operations on the different platforms. Each `memcpy()` operation copies 100 bytes of randomly initialized data between two buffers. Each sample of an exponential random variable is computed via the inverse cdf method using a `u16807` random number generator [22]. The measurements are reported in *clock ticks*.

	No. of copy operations	Block-size (in bytes)	Mean	Variance
IBM Thinkpad Linux 2.0.x	1	100	8.05	1.79
	10	10	14.37	0.62
Sun Sparc 5 Solaris 2.5.1	1	100	35.94	0.40
	10	10	293.15	21.30

(b) Our statistics are clock-tick averages (at 100 ticks/sec) over 100 samples (using UNIX `times()`) with each sample involving 200000 operations on each platform. The single `memcpy()` moves 100 bytes of randomly initialized data between two buffers. The 10 consecutive `memcpy()` operations move the same 100 bytes in ten separate steps.

Figure 2: Memory management statistics

with an equivalent sequential simulation. Time warp overheads — like state saving and rollback — make event *granularity* (i.e., an event's computational overhead) a useful measure for predicting the viability of parallel simulation for a particular model. Large event granularity is generally favorable since time warp overheads represent a smaller proportion of a model's runtime, relative to models with small event granularity. Figure 2a shows the cost of copying 100 bytes of data via `memcpy()` and the cost of computing an exponential random variable via the traditional inverse cdf method (a typical simulation event) on different platforms. These measurements show that we can save 100 bytes of simulation state in about 20% the time spent executing a simple simulation event on the test platforms.

Several approaches to saving state have been proposed in the literature. The *basic* state saving algorithm saves a processor's entire local state after each event. At rollback time, the processor's entire state is rebuilt from the appropriate checkpoint buffer. The *incremental state saving algorithm* improves upon the basic algorithm by only saving the portion of the processor state which an event changes. At rollback time, a processor undoes each incremental change to the processor state since the rollback v.t. [19, 39].

In the object based state saving algorithm, the state of each simulation object and thread changed by that event is saved after each event [29, 28]. Even if only a small portion of an object's state is changed by an event's execution, the object's entire state is saved. This potential increase in state saving overhead is offset by a decrease in state restoration overhead. At rollback time we need only discover the last buffer saved for an object before the rollback v.t. to restore the object, rather than undoing each incremental change to the object. Object based state restoration also benefits from the efficiency of single-block memory copying procedures as compared to variable-by-variable copying. For example, consider the task of saving the state of a forty-byte object consisting of ten integers. It is much more efficient, on most architectures, to save this object as a single forty-byte unit by calling `memcpy (target, object, 40)` than to save the object as ten four byte integers

via ten calls to `memcpy (target, object, 4)` (see Figure 2b).

3.1 Selective State Saving in ParaSol

ParaSol attempts to improve upon object based state saving with a *selective state saving algorithm* (SSA). Rather than saving dirty objects after each event, the SSA algorithm saves dirty objects only after several events execute. At rollback time, the simulation kernel must re–execute events which were rolled back as a result of the state saving period; this re–execution is known as *coasting forward* [24, 37].

Despite the extra cost of coasting at rollback time, selective state saving can reduce state saving overhead if the same object is changed by several sequentially executed events. In this case, the dirty object is saved only once with the selective state saving algorithm rather than multiple times with the simple object based algorithm. To balance the cost of coasting with the benefit of reduced state saving, ParaSol employs an adaptive algorithm which dynamically adjusts the size of the *state saving interval* (the number of events processed between each state saving snapshot) [28].

ParaSol's object based state saving mechanism interacts with the selective state saving algorithm in ways not anticipated by Lin [24] or Rongren [37]. Lin and Rongren analyze selective state saving with the assumption that the state of the whole system is saved at checkpoint time (our basic state saving algorithm). They therefore predict that the cost of saving state after every N events is only $1/N$ the cost of saving state after every 1 event (plus the added cost of coasting forward).

ParaSol only saves and restores the state of dirty objects at state saving and restoration time, respectively. If ParaSol saves state every N events, and each event in any set of N events touches a different object (i.e., N dirty objects), then ParaSol saves the same amount of state (N objects) whether state saving is done every N events or every 1 event. Furthermore, saving state after every event should be cheaper than saving after every N events, since coasting forward is not necessary at rollback time.

We can gain intuition into the costs and benefits of selective state saving in ParaSol by analyzing the impact of SSA on a simple model of a ParaSol simulation. We measure the benefit of SSA relative to object based state saving after each event with a benefit function $B()$. Function $B()$ is positive if SSA reduces parallel simulation runtime, it is negative if SSA increases runtime, and it is zero if SSA does not affect runtime.

Let $d(m)$ be a function that represents the number of objects *dirtied* after m events are executed. Any decrease in state saving cost that results from saving $d(m)$ objects once, instead of saving $d(1)$ objects m times, is clearly due to the SSA. The expression

$$B(m) = m * d(1) - d(m) \tag{1}$$

is the work avoided by saving $d(m)$ objects after m events instead of saving $d(1)$ objects after each event. *Coasting forward* over extra events at rollback time is a cost of SSA. Assuming that the average number of events which must coast forward at rollback time is equal to half the number of events in the state saving interval,

$$B(m, c, p_1) = m * d(1) - d(m) - p_1(m) * c * m/2 \tag{2}$$

yields a new benefit function. In this equation c is the cost of re-executing (coasting forward) an event relative to the cost of saving an object's state; $p_1(m)$ is the probability of a single rollback into a state saving interval of size m. We ignore the case where there is more than one rollback into the same state saving interval. Finally, another benefit of SSA is the state saving work we avoid when there is a rollback. For example, a processor P may execute 2 events after its last checkpoint when

its forced to rollback. The SSA algorithm allows the processor to forego the overhead of saving state after those 2 rolled back events. This is because *the state saved after each event which is rolled back is useless state.* Assuming that the average number of events executed since the last checkpoint at rollback time is equal to half the number of events in the state saving interval,

$$B(m, c, p_1, p_2) = m * d(1) - d(m) - p_1(m) * c * m/2 + p_2(m) * d(m/2) \tag{3}$$

yields a new benefit function. In this equation $p_2(m)$ is the probability of a rollback *from* a state saving interval of size m.

We make the following simplifying assumptions:

- the average cost of state saving an object is equal to the average cost of executing an event, so that $c = 1$;
- the probability of a rollback from a given state saving interval equals the probability of rolling back to a given state saving interval so that $p_1(m) = p_2(m) = p(m)$;
- each event dirties exactly one object, so that $d(1) = 1$.

With the above assumptions, the SSA benefit function can be simplified to:

$$B(m, c, p) = m - d(m) - p(m) * m/2 + p(m) * d(m/2), \tag{4}$$

which we can approximate as

$$B(m, p) = (m - d(m)) * (1 - p(m)/2). \tag{5}$$

The $(m - d(m))$ factor in the benefit function increases as the state saving interval m increases, but the $(1 - p(m)/2)$ factor decreases as m increases. Since our benefit function is not continuous, we cannot simply differentiate to find an optimal value of the state saving interval m in the general case. We can, however, consider special cases.

If we assume that $d(m)$ is a near-linear function of m, our benefit function simplifies further to:

$$B(m, p) = (m - m * d(1)) * (1 - p(m)/2) = (m - m) * (1 - p(m)/2) = 0, \tag{6}$$

which suggests that SSA has nothing to offer models which exhibit a near-linear dependence between the number of events and the number of different objects dirtied by those events, in a given state saving interval. Consider, for example, a queueing model with N simulation objects. an event may dirty each object every T time units, where T is an exponentially distributed random variable. The simulator may save state every m events (m is the state saving interval), at which point all dirty objects become clean. This model is representative of a large class of queueing network models. Our simple model may resemble (in object dirtying behavior) the simulation of a communication network where many packets (events) move about between switches (objects). Our model may also resemble the behavior of a manufacturing simulation where components (events) of a product move between different stations (objects) of an assembly line. If the number of objects N in the model is large compared to the state saving interval m, then it is unlikely that two events touch the same object in the same interval.

3.2 Lazy State Saving

In the previous section we argued that selective state saving may not improve the performance of the object based state saving algorithm for queuing models with many objects. However, our analysis of the SSA benefit function $B()$ reveals an important overhead of object based state saving,

namely, that saving state after events which are later rolled back *is wasted work*. We can try to avoid saving state that will be thrown away by a rollback by waiting as long as possible before saving a dirty simulation object's state. If rollback occurs before we actually save the dirty object, then we avoid saving the object's state. We call this the *lazy state saving* algorithm (LSA).

To support lazy state saving, a PDES system must supply a mechanism whereby a simulation object may signal the simulation kernel that the object is being accessed before the object's state changes. When the kernel receives the access signal, the kernel saves the object's state. The function $B(m, c, p_1)$ in Eq. 2 calculates the benefit of using SSA rather than LSA. Note that SSA no longer benefits from the saved cost of not saving objects dirtied by events that are rolled back, since LSA doesn't save these objects either. Assuming, as done earlier, that $d(x)$ is near-linear in x, we obtain

$$B(m, c, p_1) = -p_1(m) * c * m/2 \tag{7}$$

after simplifying our expressions. The negative result indicates that SSA adds more overhead (and thus, is of less benefit) than LSA to the simulation of a model with many objects.

3.3 Memory Management and Lazy State Saving

In addition to the advantages stated so far, lazy state saving also allows for efficient memory use. With LSA, a simulation object can simply push its checkpoint onto a stack in memory. At rollback time, each dirty object restores its state from the appropriate checkpoint, and the top of the memory stack is moved back to the last checkpoint made before the rollback destination. At fossil collection time, the bottom of the memory stack is advanced to the first checkpoint made after the new gvt. Non-lazy approaches to state saving can not exploit this stack-based approach to memory management since each object needs to keep a "last checkpoint" valid at all times.

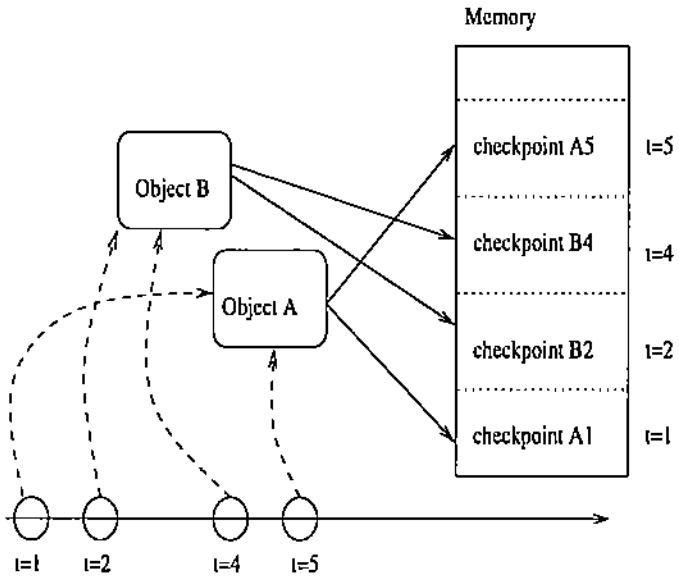
We can illustrate the difference between lazy and non-lazy state saving more clearly with an example. Suppose that an object A is dirtied by events e_1 and e_5 at virtual times 1 and 5, respectively. Using lazy state saving, A would checkpoint its state *before* being accessed by e_1, then again *before* being accessed by e_5. If there is a rollback to v.t. 3, then A restores its state from the checkpoint collected before e_5. When the simulation's global virtual time (gvt) advances to, say 4.5, then A can discard the checkpoint made before e_1, since A cannot rollback to a v.t. before 1.

Using non-lazy state saving, A checkpoints its state *after* being dirtied by e_1, and again *after* being dirtied by e_5. If there is a rollback to v.t. 3, then A discards the checkpoint made after e_5, and A restores its state using the checkpoint made after e_1. When the gvt advances to 3, A cannot discard the checkpoint made *after* e_1 since A may need this checkpoint in the event of a rollback to any v.t. before 5. Systems which use non-lazy state saving cannot make efficient use of stack based memory management since the first valid checkpoint for an object may be collected at an arbitrary virtual time.

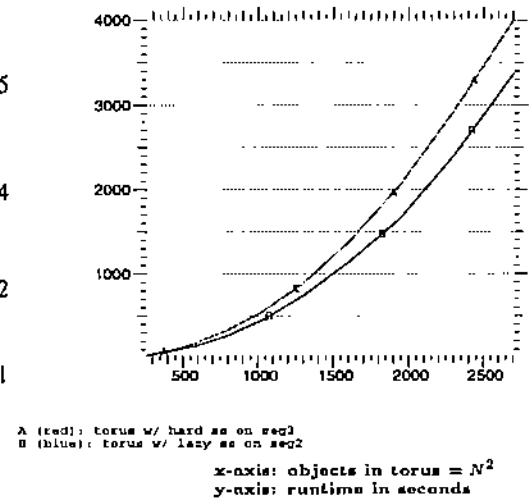
Figure 3a illustrates how memory cannot be easily reclaimed from a stack when using non-lazy state saving. We can see that events touch object A at v.t. 1 and 5, and events touch object B at v.t. 2 and 4. Using non-lazy state saving we may reclaim checkpoint $B2$ when gvt advances to 4.5, but we must keep every other checkpoint against the possibility of a rollback to some v.t. after 4.5. Using lazy state saving, we can reclaim checkpoints $A1$, $B2$, and $A4$ since they are only useful for restoring state before v.t. 1, 2, and 4 respectively.

3.4 Measurements

Most of our performance measurements record simulation runtimes for different sizes of a torus queueing model. The torus is a simple model where customers migrate randomly between neighboring servers arranged in a two dimensional mesh whose ends connect to form a torus. Each



(a) Stack memory management



(b) Torus runtimes using non-lazy (A – red) and lazy (B – blue) state saving on 2 processors of a 4 processor sparc 20 with 128 MB memory.

Figure 3: Lazy state saving

server keeps a FIFO queue of customers waiting for service. After being serviced for an exponentially distributed amount of time at a server, a customer moves to one of the server's four neighbors. The torus consists of $N \times N$ servers over which $N \times N/2$ simulation processes (threads) randomly migrate. Unless otherwise noted, our graphs plot the runtime in seconds on the y axis against the size $N \times N$ of the torus on the x axis.

Although the torus is a deceptively simple model, it is both a challenging test and is representative of other models. Thus, it is a useful tool for comparing the performance of different simulation software designs. The torus is easy to port between different simulation systems, and can be configured to vary event granularity and frequency of interprocessor communication. Most simulation researchers are familiar with the torus model so that a discussion of simulation performance can focus on issues in the design of the simulation system rather than details of the implementation of a special model. We feel that the torus and similar queueing networks are sufficiently challenging PDES applications so that if a parallel simulation system performs well on these, it is more than likely to perform well on other models.

Figure 3b compares the runtimes of two-processor torus simulations that use non-lazy and lazy state saving. The simulations using lazy state saving run about 10% faster than the non-lazy simulations.

4 Laziness and the Simulation Calendar

A simulation calendar should allow a simulation to efficiently schedule future events and determine the next event to execute. In a thread based simulation language like ParaSol, each simulation event corresponds to a thread resumption, so that the simulation calendar turns into a scheduling queue for the thread system.

4.1 Sequential Simulation Calendar

In sequential simulators, the simulation calendar must support two basic operations:

- `schedule-event` — schedule an event in the simulation calendar for execution at some virtual time $t_e \geq t_c$, where t_c is the current virtual time of the system.
- `get-next-event` — get the next event for the simulation to execute and advance the system's virtual time to $t_c = t_e$.

Each event simulated involves one `schedule-event` and one `get-next-event` calendar operation.

Various efficient calendar implementations have been employed in sequential simulation languages. Most calendar data structures implement some kind of heap based priority queue (a notable exception is the *calendar queue* [9]). Heap based priority queues support the `get-next-event` and `schedule-event` operations in $O(\log n)$ operations (where n is the size of the calendar) [14, 13, 42].

An efficient calendar implementation is essential for achieving good runtime performance by a simulator. A doubly linked list (dll) based calendar implements the `schedule-event` operation in $O(n)$ time and the `get-next-event` operation in $O(1)$ time. Therefore, each simulated event requires $O(n)$ time in a dll calendar. Similarly, each simulated event requires only $O(\log n)$ time in a heap based calendar. As the size of the simulation and the size of the calendar grow, so too grows the ratio by which a simulator with a heap calendar is faster than a simulator with a dll calendar.

4.2 Time Warp Calendar

In a time warp based parallel simulator, the calendar must implement `rollback` and `fossil-collect` methods in addition to a sequential calendar's methods (`schedule-event`, `get-next-event`). The `rollback` method restores the calendar to a state in the simulation's virtual past. The `fossil-collect` method informs the calendar that the simulation's gvt has advanced to a new value t_{gvt}. Knowing that gvt has advanced, the calendar can discard events kept in the calendar against the possibility of rollback to a state before the new gvt.

We cannot rollback the state of the calendar the same way we restore a simulation object's state. A simulation object restores itself to a state at v.t. t from a buffer saved at this time t. When the calendar rolls back, it must remember any additions made to its state as the result of messages received from other processors. So, when the calendar rolls back to a v.t. t_r from a current v.t. t_c, the calendar must remember each event scheduled by remote processors while the local simulation processed events scheduled between t_r and t_c.

To the best of our knowledge, *SPEEDES* and *APOSTLE* are the only PDES systems using a calendar specially designed for use in parallel simulation [42]. Their calendar is designed for use with a semi-conservative breathing-time-buckets based simulation. Other parallel simulation systems modify sequential calendar data structures for parallel simulation.

The *WARPED* [26] PDES system uses a doubly linked list (dll) based calendar. A dll calendar implements an m event rollback in $O(m)$ operations. Each event e in the calendar maintains a list of the events scheduled in the calendar as a result of e's execution. When a simulation rolls back from v.t. t_c to v.t. t_r, the simulation kernel unexecutes (undoes the execution of) each calendar entry (event) e with time stamp t_e, where $t_r \leq t_e \leq t_c$. The calendar unexecutes an event e by removing from the calendar every event scheduled during e's execution and sending anti-messages to cancel every event which e scheduled on remote processors [20]. Since the calendar must unexecute each event that is rolled back, we know that $O(m)$ is a lower bound on the performance of the `rollback-to-vt` method. Finally, a dll calendar performs the `fossil-collect` method in $O(n)$ steps by searching for the last calendar entry e executed with time stamp $t_e | t_e < t_{gvt}$ and discarding the calendar entries before e inclusive.

The *GTW* PDES system bases its calendar on a heap. A heap supports `schedule-event` and `get-next-event` in $O(\log n)$ steps. Unfortunately, keeping every valid event in a heap means that the `rollback` and the `fossil-collect` methods require $O(n * \log n)$ steps in order to maintain the balanced tree structure of the calendar.

4.3 The Lazy Hybrid Time Warp Calendar

To begin, we attempt to combine the good sequential performance of the heap calendar with the good time warp performance of the dll calendar by letting the calendar keep *already executed* events in a doubly linked list while maintaining *pending* events in a heap. This *hybrid calendar* can implement `fossil-collect` in $O(n)$ time since this method only involves the *already executed* events in the doubly linked list part of the calendar. Table 1 compares the hybrid calendar's performance with the other calendars. PARASOL uses a tree rather than a heap based calendar since the tree supports a `search` method in $O(\log n)$ operations [14]. The `search` operation is useful for finding an entry canceled by an anti-message.

Operation	List	Heap	Hybrid
`schedule`	$O(n)$	$O(\log n)$	$O(\log n)$
`get-next`	$O(1)$	$O(\log n)$	$O(\log n)$
`rollback`	$O(m)$	$O(m * \log n)$	$O(m * \log n)$
`fossil`	$O(q)$	$O(q * \log n)$	$O(q)$

Table 1: Calendar performance in optimistic simulation.

We can improve the performance of our hybrid calendar even further if we allow our calendar to be lazy. When a new event e_1 is scheduled, we simply place e_1 at the end of a list of *potential* events. The potential events are not placed in the calendar until the v.t. of the next event in the calendar exceeds the v.t. of the earliest potential event. At this *crossover* point, every event in the potential list is inserted in the calendar. If a rollback occurs before the simulation reaches the crossover point, then we can "unschedule" potential events by simply dropping them off of the end of the potential list. In this way we avoid the overhead of scheduling events which are later unscheduled by a rollback. Furthermore, if every event unscheduled by a rollback is in the potential list, then we do not rebalance the calendar's tree at rollback time. Figure 4 depicts the lazy data structure along with a table of its runtime characteristics.

4.4 Measurements

We use the torus model (see Section 3.4) to evaluate the impact of our different calendar designs on parallel simulation performance. Some previous work comparing the performance of different simulation calendars base their results on calendar performance under different synthetic work loads. These synthetic work loads do not lend themselves well to the study of calendar performance in parallel simulation since they do not take rollback into account. Rather than attempt to devise a synthetic work load which may or may not accurately model the stresses on a parallel simulation's calendar, we simply collect measurements from simulations of the torus.

Figure 5 shows simulation runtimes for different sizes of the torus model running on Purdue's SP2 using 1, 2, 4, and 6 processors. We can see that the simulations which use the lazy calendar (B – blue lines) consistently outperform the simulations using a list based calendar (A – red lines). As the number of processors participating in the simulation increases, the benefit of using the lazy calendar decreases because the size of the calendar on each processor gets smaller as more processors

Operation	Best Case	Worst Case
`schedule`	$O(1)$	$O(1)$
`get-next`	$O(\log n)$	$O(p * \log n)$
`rollback`	$O(m)$	$O(m * \log n)$
`fossil`	$O(q)$	$O(q)$

- p – number of events in the potential list
- m – number of events rolled back
- q – number of events fossil collected

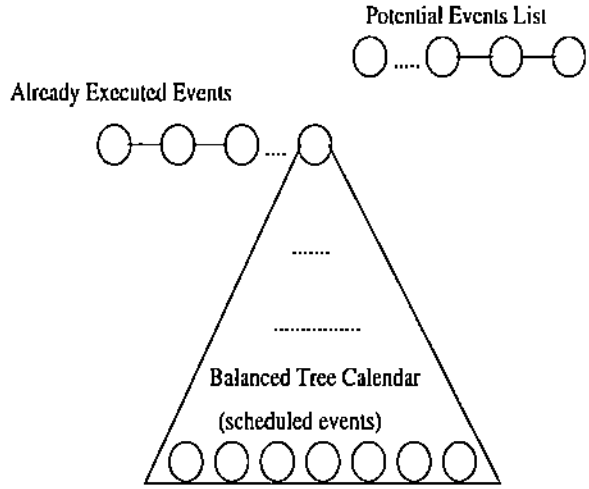
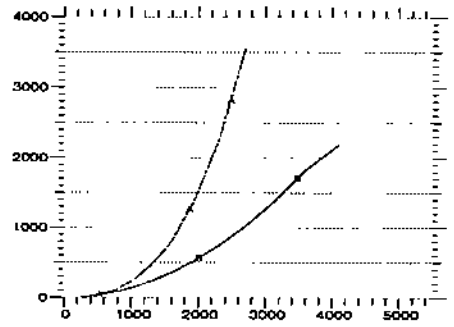


Figure 4: Lazy calendar

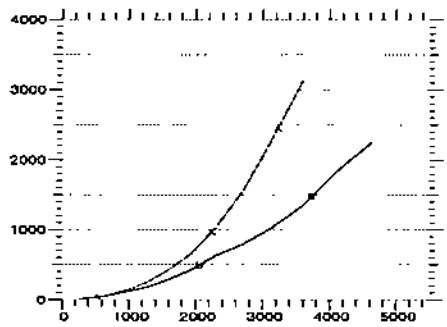
are used. The dll calendar performs well when there are few entries in the calendar, while the tree based calendar shows the most benefit when there are many entries in the calendar.

It is not surprising that a tree based calendar should perform better than a list based calendar in a sequential simulation. In a parallel simulation, however, the tree based calendar is expensive to roll back. As explained earlier, our lazy calendar tries to lower rollback costs by scheduling events in the tree data structure as late as possible. Figure 6 shows some measurements comparing the performance of a lazy hybrid calendar (which delays inserting events into the calendar's tree) to the performance of a non-lazy hybrid calendar. The results show that the lazy calendar performs measurably better than the non-lazy calendar in the simulations using only two processors (Figure 6a). However, as we increase the number of processors, the benefit of using the lazy calendar decreases (Figure 6b) and then disappears (Figure 6c). This results because, as the number of processors increases, the size of of the calendar on each processor decreases; so an efficient calendar implementation has a smaller impact on overall performance. Also, for our torus model, the rollback frequency decreases as the size of the model increases (see below), so that the importance of avoiding work at rollback time becomes less important. From these results we may conclude that a lazy calendar can benefit simulation performance in a simulation with a large calendar and frequent rollback, and a lazy calendar will not hurt performance in other circumstances.

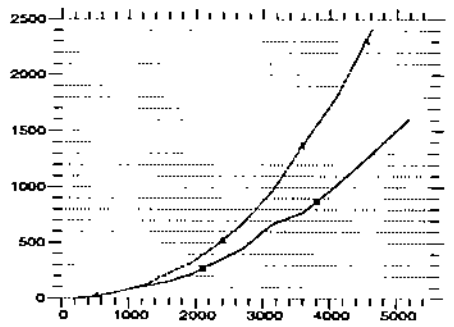
Figure 7 plots the average rollback period (number of events processed between consecutive rollbacks), rollback size (number of events rolled back), and lazy list size (the number of events in the lazy potential list at rollback time) for a series of 4 processor torus simulations on the Purdue SP2. These plots give us some intuition into the work which the lazy calendar avoids. We can see that as the size of the model increases, the values of our three variables (rollback period, rollback size, and lazy list size) increase, but Figure 7d shows us that the rollback period increases much more quickly than the rollback size or the lazy size. This makes sense when we consider that the transaction density (number of events per unit virtual time) increases faster than the rate of interprocessor communication as the size of the torus model increases. For an $N \times N$ torus, the transaction density is proportional to N^2 (the number of objects and threads simulated on each processor), while the rate of interprocessor communication is proportional to N (the number of objects along the boundary separating each processor). Since interprocessor communication is the ultimate cause of rollback, we are not surprised to find that rollback is less frequent (as a function of the number of events processed between rollbacks) in larger simulations.



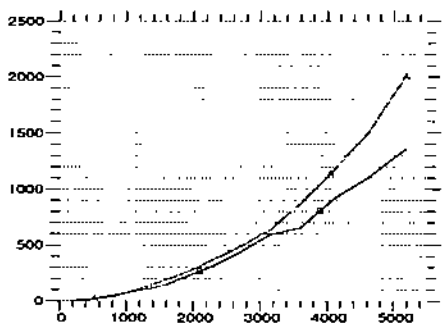
(a) Lazy (B-blue) vs. list (A-red) calendar, 1 processor



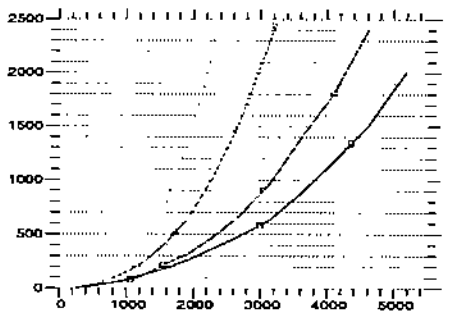
(b) Lazy (B-blue) vs. list (A-red) calendar, 2 processors



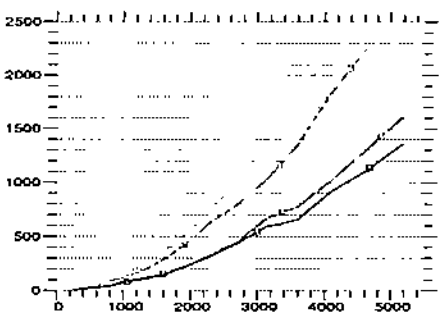
(c) Lazy (B-blue) vs. list (A-red) calendar, 4 processors



(d) Lazy (B-blue) vs. list (A-red) calendar, 6 processors



(e) List calendar performance (yellow - 1p, green - 2p, red - 4p, blue - 6p)



(f) Lazy calendar performance (yellow - 1p, green - 2p, red - 4p, blue - 6p)

x-axis: objects in torus $= N^2$
y-axis: runtime in seconds

Figure 5: Torus runtimes with doubly linked list and lazy calendar on SP2

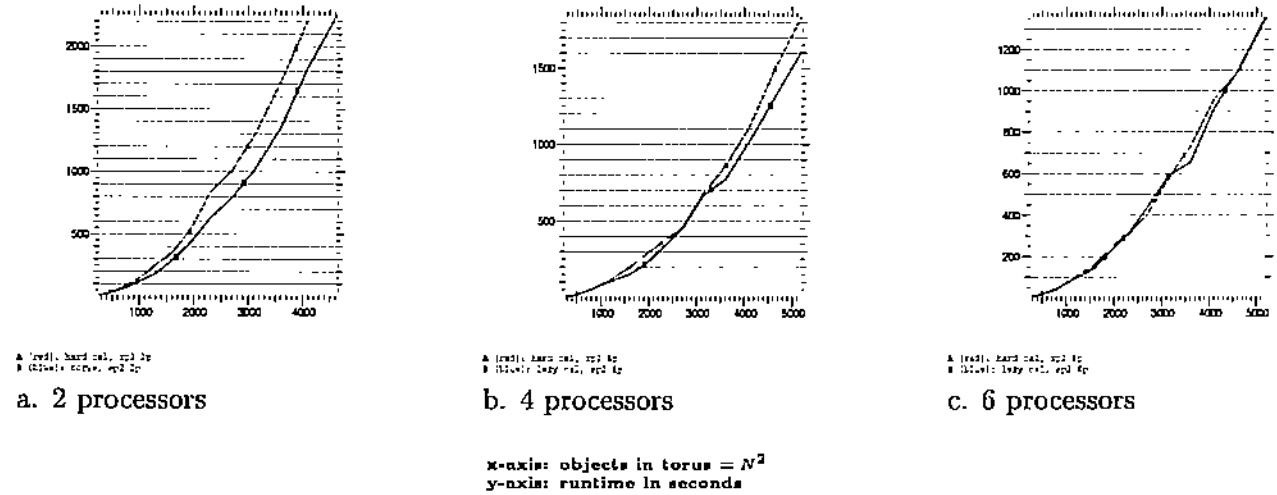


Figure 6: Hybrid calendar (A-red) vs. lazy hybrid calendar (B-blue)

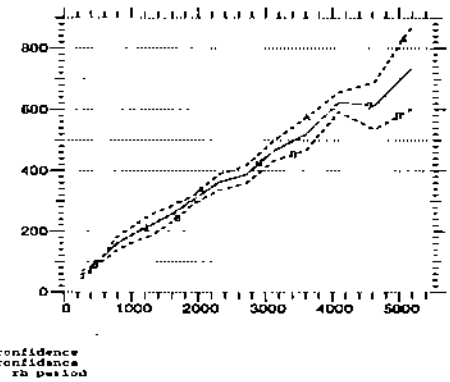
5 Laziness and Lookahead

The lazy object state saving algorithm (see Section 3.2) and the lazy hybrid calendar both exploit the *lookahead* [23, 41] available in a discrete event simulation to avoid work (state saving and scheduling) which may be rolled back. The larger an application's lookahead, the more successful lazy algorithms are at avoiding wasted work.

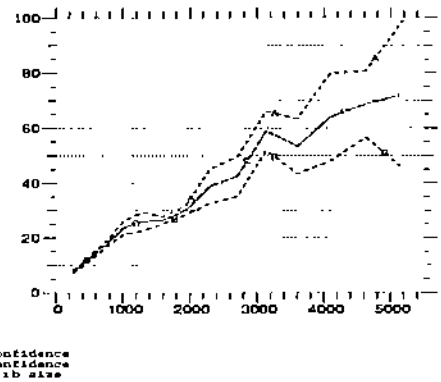
When implementing the lazy calendar we discovered lookahead destroying *zero lookahead* events in ParaSol's time warp kernel. A zero lookahead event e_0 is scheduled at the same virtual time as the event that creates e_0. For example, when a ParaSol thread T_a creates a new thread T_b, T_a schedules a zero lookahead event in the calendar to begin T_b's execution at the *current* virtual time. A zero lookahead event also results when we resume a thread that is taken off of a wait queue. For example, a facility may be servicing one customer (thread) T_a while another customer T_b waits in a queue. When T_a's (virtual) time at the facility expires, T_a *releases* the facility to T_b. Thread T_a resumes T_b by removing T_b from the facility's queue and scheduling a zero lookahead event to execute T_b at the current virtual time.

A zero lookahead event degrades the performance of our lazy calendar. When an event e_a schedules a zero lookahead event e_b, the calendar flushes the list of potential events. Recall that the lazy calendar places newly scheduled events onto a list of potential events. The calendar does not insert potential events into the calendar's heap unless the next scheduled event is in the potential list. The goal in keeping the potential list is to delay scheduling events which are later descheduled by a rollback (see Section 4.3). A zero lookahead event e_b forces the calendar to flush its list of potential events since e_b executes immediately after the current event e_a (which scheduled e_b).

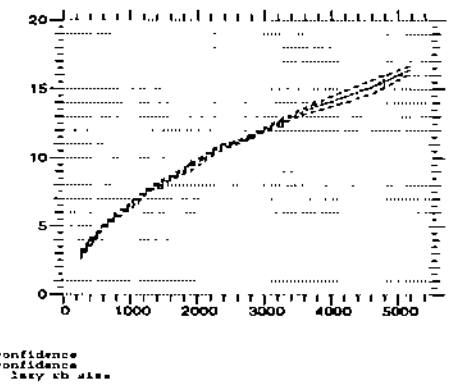
In ParaSol we avoid the negative effects of zero lookahead events by treating them as a special case. When the current event e_a generates a zero lookahead event e_b to execute at the current virtual time, the ParaSol kernel does not insert e_b into the calendar. Rather, ParaSol's kernel simply appends e_b onto a list of zero lookahead events which the kernel automatically executes in order, after the current event finishes. This solution has two benefits. First, we avoid the overhead of scheduling a zero lookahead entries in the calendar. Scheduling a zero lookahead event e_b is wasted



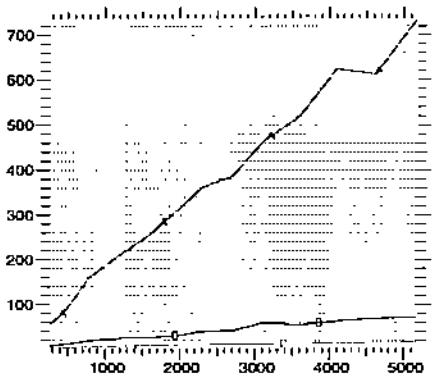
(a) average rollback period (events between rollbacks) with confidence interval



(b) average rollback size (events rolled back) with confidence interval



(c) average number of events in calendar potential list at rollback time



(d) (A,red) - rb period, (B,blue) - rb size, (C,green) - potential size

x-axis: objects in torus = N^2
y-axis: number of events

Figure 7: Simulation time warp statistics

effort since we know that e_b should run after the current event completes execution. Second, we avoid the overhead of removing zero lookahead entries from the calendar if a rollback occurs. We call this technique of executing multiple events for each calendar access *event grouping*.

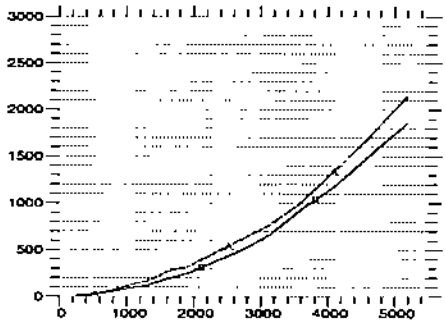
5.1 Aggressive Migration

A different kind of zero lookahead event occurs when a thread migrates. When a thread T_a on processor P_0 attempts to access an object located on a remote processor P_1, the PARASOL kernel enables thread T_a to migrate to P_1 and schedules an event e_1 to resume T_a's execution on P_1. A drawback to this migration mechanism is that a thread which leaves processor P_0 at virtual time t_a executes on processor P_1 at the same v.t. t_a. Thus there is no virtual "lag time" which the thread may consume as it migrates.

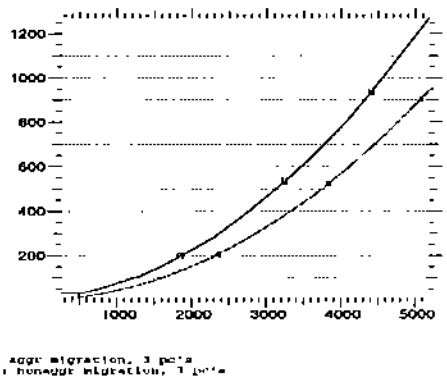
We may prefer a situation where a thread T_a which migrates from processor P_0 at v.t. t_a is scheduled to run on the destination processor P_1 at v.t. $t_a + \delta$. The delay δ may improve parallel performance in two ways. First, the thread migration is less likely to be a rollback causing straggler on P_1 since the thread is migrating into the virtual future. Even if the migration does cause P_1 to roll back, this rollback may be a good thing since it prevents P_1 from racing too far ahead of P_0 in virtual time. The second benefit a δ in migration gives us is that if processor P_0 rolls back and has to retract thread T_a's migration with an anti-message x_a, then x_a has a better chance of reaching processor P_1 before T_a executes on P_1. In other words, it's easier for an anti-message to "catch" the events which it is sent to cancel.

We added the `hold_then` primitive to the PARASOL API to support *aggressive migration* in models which do not involve preemption at the boundaries between processors. Aggressive migration simply allows a thread to migrate to the processor hosting the next object which the thread will access before the thread has relinquished a hold on its current object. The graphs in Figure 8 show us that aggressive migration measurably improves the runtime of simulations of the torus model, especially for the parallel simulations using PC's connected by the slow (compared to the SP2's interconnection network) Ethernet network.

Figure 9 shows how we use the `hold_then` primitive to improve the performance of our torus queueing network model. We can see that each job thread T_a migrates between servers arranged in an $N \times N$ torus. At each server T_a first reserves the server. If the server is not free, then T_a is placed on a queue. When T_a finally acquires the server it holds the server for a specified amount of (virtual service) time. Finally, T_a releases the server and moves on to the next server. This reserve-hold-release pattern of control flow is a key characteristic of tele-traffic models. If thread T_a's service cannot be preempted, then T_a can aggressively migrate to its next server rather than wait idly to simulate T_a's service time t_s. Furthermore, if the server serves customers in FIFO order, then the server can precompute that amount of time t_q which T_a must spend waiting in the queue. Now T_a can aggressively migrate to the processor hosting T_a's next server with $\delta = t_q + t_s$.



(a) torus runtimes on 4 processors of SP2,
(A,red) – nonaggressive migration,
(B,blue) – aggressive migration



(b) torus runtimes on 3 PC's over 10Mb/s ethernet,
(A,red) – aggressive migration,
(B,blue) – nonaggressive migration

x-axis: objects in torus = N^2
y-axis: runtime in seconds

Figure 8: Aggressive migration improves performance

```
void
Job_thread::run ()
{
 const int i_num_visits = oi_x_size * oi_y_size;
 int i_current_station = oi_initial_station;

 for ( int i = 0; i < i_num_visits; ++i ) {
   op_torus[ i_current_station ]->reserve ();
   double d_holdtime = op_rng->expon ( 10.0 );
   gp_kernel->hold ( d_wait_time );
   op_torus[ i_current_station ]->release ();
   i_current_station = choose_next_station
          ( i_current_station, op_rng->u01 () );
 }}
```

(a.) standard code

```
void
Job_thread::run ()
{
 const int i_num_visits = oi_x_size * oi_y_size;
 int i_current_station = oi_initial_station;

 for ( int i = 0; i < i_num_visits; ++i ) {
   double d_holdtime = op_rng->expon ( 10.0 );
   int    i_next_station = choose_next_station
           ( i_current_station, op_rng->u01() );
   double d_wait_time =
            op_torus[ i_current_station ]->
                   service_for ( d_holdtime );
   gp_kernel->hold_then ( d_wait_time,
                  op_torus[ i_next_station ] );
   i_current_station = i_next_station;
 }}
```

(b.) aggressive code

Figure 9: Aggressive migration in ParaSol

5.2 Threads, Events, and Calendar Entries

In ParaSol, the simulation driver acts as a thread scheduler and the simulation calendar is a thread scheduling queue. The use of threads in ParaSol yields the valuable benefit of eliminating time warp thrashing [16, 25]. When interdependencies between processes that participate in a parallel simulation produce recurring rollbacks, the simulation's forward progress is significantly slowed. This is called Time warp *thrashing*. Consider a simple form of thrashing which can occur in a three processor parallel simulation.

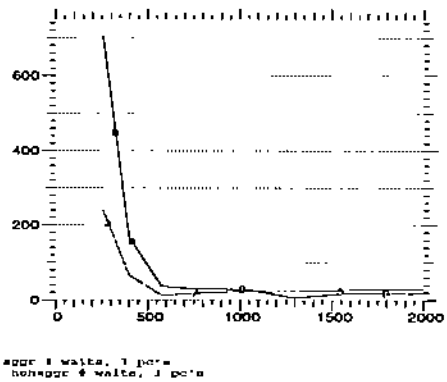
1. Processor P_1 generates an event e_1 at v.t. t_1 to be executed on processor P_2.
2. Processor P_2 generates an event e_2 at v.t. t_2 to be executed on processor P_3.
3. Processor P_3 generates an event e_3 at v.t. t_3, $t_3 < t_1 < t_2$, to be executed on processor P_1.

We can see that events travel in a circle between the processors so that an anti-message cannot "catch up" with the event it created until after that event has created another event which must be trailed by yet another anti-message.

This is only the simplest form of *time warp thrashing* which can plague a parallel runtime system. Fortunately, our use of the active-process world view helps make ParaSol invulnerable to thrashing. By associating each simulation event with a unique simulation process (thread), ParaSol can ensure that *no more than one* instance of a given thread is resident in a particular processor. Lets consider how the ParaSol runtime system behaves in the situation described above by tracking the execution of the thread T_2 associated with event e_2. Thrashing may result when processor P_2 sends an anti-message a_2 to processor P_3 to recall the migration of T_2 (execution of e_2). By the time the anti-message reaches P_3, thread T_2 has already migrated on to processor P_1 (moving in a circle). So processor P_3 sends an anti-message to processor P_1. In an event–based simulation, this process continues forever. However, in ParaSol thread T_2 will "catch up with itself" so that eventually the instance of thread T_2 being chased by an anti-message will arrive at the processor where the correct version of T_2's context is resident. When this happens, the ParaSol runtime at that processor realizes that an anti-message is pending and waits.

Figure 10 compares the number of times a PC processor must wait for an anti-message to cancel a duplicate thread when using aggressive migration (the red line labeled A) and non-aggressive migration (the blue line labeled B). The y-axis tracks the number of times the processor blocks waiting for an anti-message to destroy a duplicate thread, and the x-axis tracks the size of the model. This data was collected on a network of three PC's connected by 10 Mb/s switched ethernet. Each data point is the average of 10 simulation runs with negligible variance. We can see that as the size of the model increases, the processor blocks fewer times. As a model's size increases, the number of servers on each processor increases, so that each thread migrates less frequently to access remote objects. For the small models, we can see that the processor blocks much less often to wait for anti-messages when the simulation uses aggressive migration. This indicates that aggressive migration helps avoid thrashing.

Figure 11 displays more impressive results comparing the number of rollbacks processed by parallel simulations using aggressive and nonaggressive migration. These results were also collected over the network of three PC's connected by a 10 Mb/s ethernet. The graphs track the total number of rollbacks processed by the simulation, and the number of rollbacks caused by anti-messages. A rollback on processor P_0 caused by an anti-messages from processor P_1 is the result of a rollback on P_1. Each data point is the average of 10 simulation runs with negligible variance. We can see that aggressive migration significantly reduces the total number of rollbacks and virtually eliminates rollbacks caused by anti-messages for large simulations. In fact, the total number of rollbacks for



- **x-axis: objects in torus = N^2**
- **y-axis: # of times processor 0 blocks waiting for an anti-message to cancel a duplicate thread**

The bottom red line (labeled A) was collected using aggressive migration, the top blue line (B) was collected without aggressive migration.

Figure 10: Waiting for anti-messages avoids thrashing

the simulations using aggressive migration is less than the number of rollbacks that are caused by anti-messages in the simulations not using aggressive migration.

6 Lazy GVT

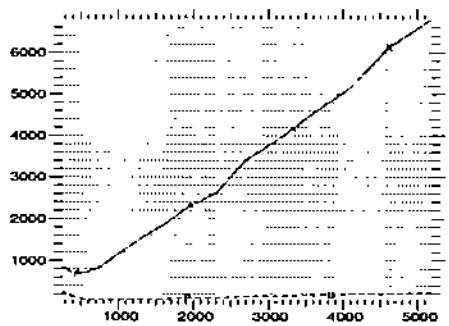
An optimistic parallel simulation periodically computes its *global virtual time* (gvt) to allow processors to *fossil collect* (i.e., reclaim) memory used to save state. A parallel simulation's gvt is the minimum of the local virtual time on each processor and all time stamps on messages in transit between processors. Good distributed algorithms exist for computing gvt [31, 11], but determining an appropriate frequency for gvt computation is still an open problem.

To arrive at a good frequency for gvt computation, a parallel simulation must address a bandwidth vs. memory trade off. If the frequency of gvt computation is too high, then the network becomes congested with gvt message traffic. If the frequency of gvt computation is too low, then simulation performance may degrade because the simulation consumes the memory resources with checkpoint buffers that evade fossil collection.

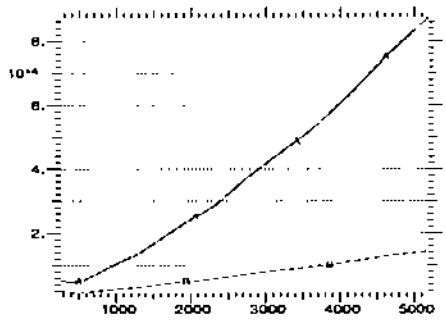
A processor P_0 initiates a gvt computation by broadcasting a message to every processor participating in the simulation. Once this is done, P_0 continues simulating events. When P_0 receives a report of every processor's local virtual time, and when all outstanding messages have been accounted for (see [31, 11]), P_0 then broadcasts a message telling every processor the new gvt.

ParaSol uses a lazy algorithm to schedule gvt computation. A processor P_0 does not initiate a gvt computation unless more than half of P_0's state-saving stack has been consumed, or P_0's local virtual time exceeds the simulation's termination time. Figure 12a shows that parallel simulations using lazy gvt scheduling outperform parallel simulations which initiate a new gvt computation immediately after the last computation completes. Figure 12b shows that the lazy algorithm computes gvt approximately every 14000 events while the aggressive scheduler computes gvt approximately every 500 events. The lazy simulation puts much less effort into gvt computation than the aggressive simulation.

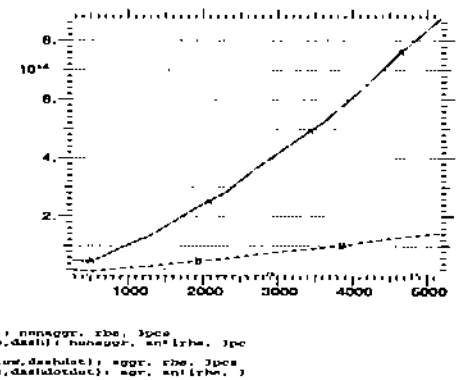
Although these initial results support lazy gvt scheduling, we must take into account the conditions under which the algorithm was tested. First, each processor used in our experiments has an abundant supply of memory (at least 128 Megabytes). Therefore, our tests did not suffer a significant performance loss by consuming more memory. Second, our experiments with ParaSol were conducted on distributed memory platforms where processors communicate by message passing.



(a) aggressive migration:
(A, solid red line) – total rollbacks,
(B, dashed blue line) – rollbacks due to anti-messages



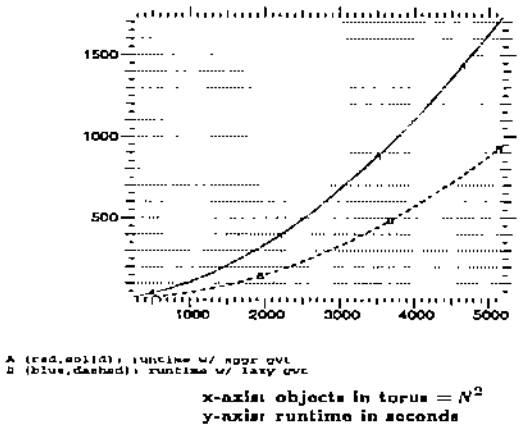
(b) not using aggressive migration:
(A, solid red line) – total rollbacks,
(B, dashed blue line) – rollbacks due to anti-messages



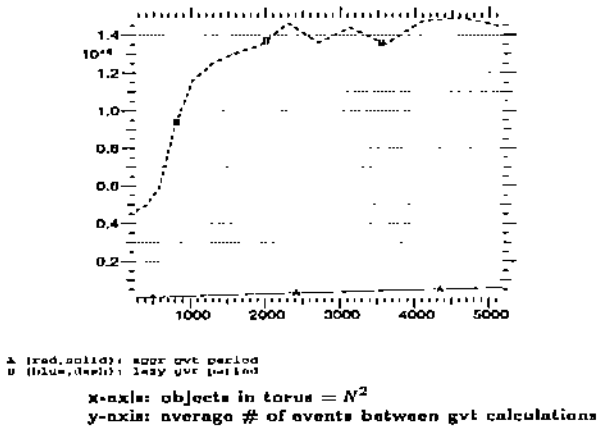
(c) graphs (a) and (b) above on the same scale

x-axis: objects in torus = N^2
y-axis: number of rollbacks

Figure 11: Aggressive migration and rollback



(a) torus runtime on 3 PC's
... (dashed line) – lazy gvt
... (solid line) – aggressive gvt



(b) gvt period on 3 PC's
... (dashed line) – lazy gvt
... (solid line) – aggressive gvt

Figure 12: Lazy gvt scheduling and performance

Efficient use of network bandwidth on these platforms (especially the network of workstations) is critical to achieving good simulation performance. The performance of parallel simulation systems which run on shared memory computer systems may not need to so carefully consider interprocessor communication overheads. However, shared memory computer systems are more expensive and less common than distributed memory systems like workstation networks.

7 Conclusions

An important source of overhead in optimistic PDES is the work which must be undone when a processor rolls back. When a processor P rolls back from vt t_{now} to vt t_{old}, then all the work which P carried out to advance simulation time from vt t_{old} to t_{now} is thrown away. We proposed lazy techniques for saving state and calendar management in an attempt to avoid doing work which is thrown away as the result of a rollback.

The lazy state saving algorithm does not save a dirty object's state until just before the object is next accessed. If the object's host processor rolls back before an event accesses the dirty object, then the object rolls back its local state and avoids collecting a checkpoint that would have been thrown away. Lazy sate saving performs better than selective state saving on many types of models by avoiding saving state thrown away by a rollback without the need to coast forward after a rollback.

The lazy hybrid calendar combines the good sequential properties of a heap with the good time warp properties of the doubly linked list by keeping pending events in a heap and already executed events in a list. The lazy calendar also appends newly scheduled events to a potential list to avoid scheduling events which are later unscheduled by a rollback. Events in the potential list are not inserted into the calendar's heap unless the earliest event in the potential list is the next event scheduled for execution. When a rollback occurs, it is much cheaper to remove unscheduled events from the potential list than from the heap.

Both lazy state saving and the lazy calendar exploit a model's lookahead to defer as long

as possible doing work which may be rolled back. PARASOL uses event grouping and aggressive migration to help exploit the lookahead present in a model. Event grouping simply allows PARASOL to treat multiple events scheduled at the same virtual time as a single event. Event grouping improves simulation performance by slowing the growth of the calendar and allowing objects dirtied by a sequence of consecutive events to avoid saving state after each event. Aggressive migration allows a thread to migrate to a processor hosting a remote object before the thread is ready to access the object. Aggressive migration helps avoid time warp thrashing by allowing virtual time to advance while a thread migrates.

Finally, lazy gvt scheduling defers gvt computation until a simulation's memory resources begin running low. Gvt computation stresses a computer network with all-to-one and one-to-many communication. Lazy scheduling helps limit communication congestion in a parallel simulation by conservatively scheduling gvt computation only when fossil collection becomes necessary.

By using lazy approaches to optimistic simulation we have significantly improved the performance of the PARASOL PDES system. We believe that, with further improvements in performance and ease of use, parallel simulation systems will become a useful tool for simulation users.

References

[1] S. Aji, A. C. Palaniswamy, and P. A. Wilsey. Interactions of optimizations to a time warp synchronized digital system simulator. *Modeling and Simulation*, pages 593–597, June 1993.

[2] D. Baezner, G. Lomow, and B. Unger. Sim++: The transition to distributed simulation. In *Distributed Simulation*, volume 22 of *SCS Simulation Series*, pages 211–218, January 1990.

[3] R. L. Bagrodia and Wen-Toh Liao. Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of SCS Multiconference on Distributed Simulation*, pages 205–210, 1990.

[4] D. Ball and S. Hoyt. The adaptive time-warp concurrency control algorithm. In *Proceedings of the SCS MultiConference on Distributed Simulation*, pages 174–177, 1990.

[5] S. Bellenot. Global virtual time algorithms. In *Proceedings of the Multiconference on Distributed Simulation*, volume 22, pages 122–127, 1990.

[6] C. J. M. Booth and D. I. Bruce. Stack-free process-oriented simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 182–185, 1997.

[7] A. Boukerche and C. Tropper. A static partitioning and mapping algorithm for conservative parallel simulations. In *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*, pages 164–172, 1994.

[8] Azzedine Boukerche and Carl Tropper. Semi-global time of the next event algorithm. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 68–77, 1996.

[9] R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31:1220–1227, October 1988.

[10] Doug Burger and James R. Goodman. Billion-transistor architectures. *Computer*, pages 46–50, September 1997.

[11] K. Chandy and L. Lamport. Distributed snapshots: Determining the global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[12] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[13] K. Chung, J. Sang, and V. Rego. A performance comparison of event calendar algorithms: An empirical approach. *Software–Practice & Experience*, 23(10):1107–1138, Oct. 1993.

[14] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. Massachusetts Institute of Technology, 1990.

[15] Phillip Dickens, Paul Reynolds, and Mark Duva. Analysis of bounded time warp and comparison with yawns. *ACM Transactions on Modeling and Computer Simulation*, October 1996.

[16] Robert Felderman and Leonard Kleinrock. Two processor time warp analysis: Some results on a unifying approach. *International Journal of Computer Simulation*, 1990.

[17] J. Fleischmann and P. A. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58, 1995.

[18] R. Fujimoto. Time warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.

[19] F. Gomes, B. Unger, and J. Cleary. Language based state saving extensions for optimistic parallel simulation. In *Proceedings of the 1996 Winter Simulation Conference*, 1996.

[20] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[21] F. Knop. *Software Architectures for Fault-Tolerant Replications and Multithreaded Decompositions: Experiments with Practical Parallel Simulation.* PhD thesis, Department of Computer Sciences, Purdue University, 1996.

[22] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 1982.

[23] Y. B. Lin and E. D. Lazowska. Exploiting lookahead in parallel simulation. Technical Report 89-10-06, Dept. of Computer Science, Univ. of Washington, 1989.

[24] Y. B. Lin, B. R. Priess, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in time warp simulations. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, volume 23, pages 3–8, 1993.

[25] Boris Lubachevsky and Alan Weiss. An analysis of rollback–based simulation. *ACM Transaction on Modeling and Computer Simulation*, 1(2):154–193, 1991.

[26] Dale E. Martin and Timothy J. McBrayer. *Warped – A TimeWarp Parallel Discrete Event Simulator (Documentation for version 0.8).* Dept. of ECECS, University of Cincinnatti, Computer Architecture Design Laboratory, Dept of ECECS, PO Box 210030 Cincinnati, OH 45221-0030, December 1997.

[27] E. Mascarenhas. *A System for Multithreaded Parallel Simulation and Computation with Migrant Threads and Object.* PhD thesis, Department of Computer Sciences, Purdue University, 1996.

[28] E. Mascarenhas, F. Knop, R. Pasquini, and V. Rego. Checkpoint and recovery methods in the parasol simulation system. In *Proceedings of the 1997 Winter Simulation Conference*, pages 452–459, 1997.

[29] E. Mascarenhas, F. Knop, and V. Rego. ParaSol: A Multi-threaded System for Parallel Simulation Based on Mobile Threads. In *Proceedings of the Winter Simulation Conference*, pages 690–697, 1995.

[30] Edward Mascarenhas, Felipe Knop, and Vernon Rego. Minimum cost adaptive synchronization: Experiments with the parasol system. In *Proceedings of the 1997 Winter Simulation Conference*, pages 389–396, 1997.

[31] Friedmann Mattern. Efficient algorithms for distributed snapshot and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[32] David Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *Parallel Programming: Experience with Applications, Languages, and Systems*, July 1988. ACM SIGPLAN.

[33] Kiran S. Perumalla and Richard M. Fujimoto. Adaptive flow control in time warp. In *11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 108–115, June 10-13 1997.

[34] Tekla S. Perry. In search of the future of air traffic control. *IEEE Spectrum*, pages 18–35, August 1997.

[35] P. Reiher. Parallel simulation using the time warp operationg system. In *Proceedings of the Winter Simulation Conference*, pages 38–45, 1990.

[36] P. Reiher and D. Jefferson. Limitation of optimism in the time warp operating system. In *Proceedings of the Winter Simulation Conference*, pages 765–769, 1989.

[37] R. Rongren and R. Ayani. Adaptive checkpointing in time warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 94–100, 1994.

[38] H. D. Schwetman. CSIM: A C-based process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.

[39] J. Steinman. Incremental state saving in SPEEDES using C++. In *Proceedings of the Winter Simulation Conference*, pages 687–696, December 1993.

[40] J. S. Steinman. SPEEDES: A unified approach to parallel simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, volume 24 of *Simulation Series*, pages 75–84, 1992.

[41] J. S. Steinman. Breathing Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 109–118, 1993.

[42] J. S. Steinman. Discrete-event simulation and the event horizon part 2: Event list management. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 170–178, 1996.

[43] P. Wonnacott and D. Bruce. The apostle simulation language: Granularity control and performance data. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, 1996.